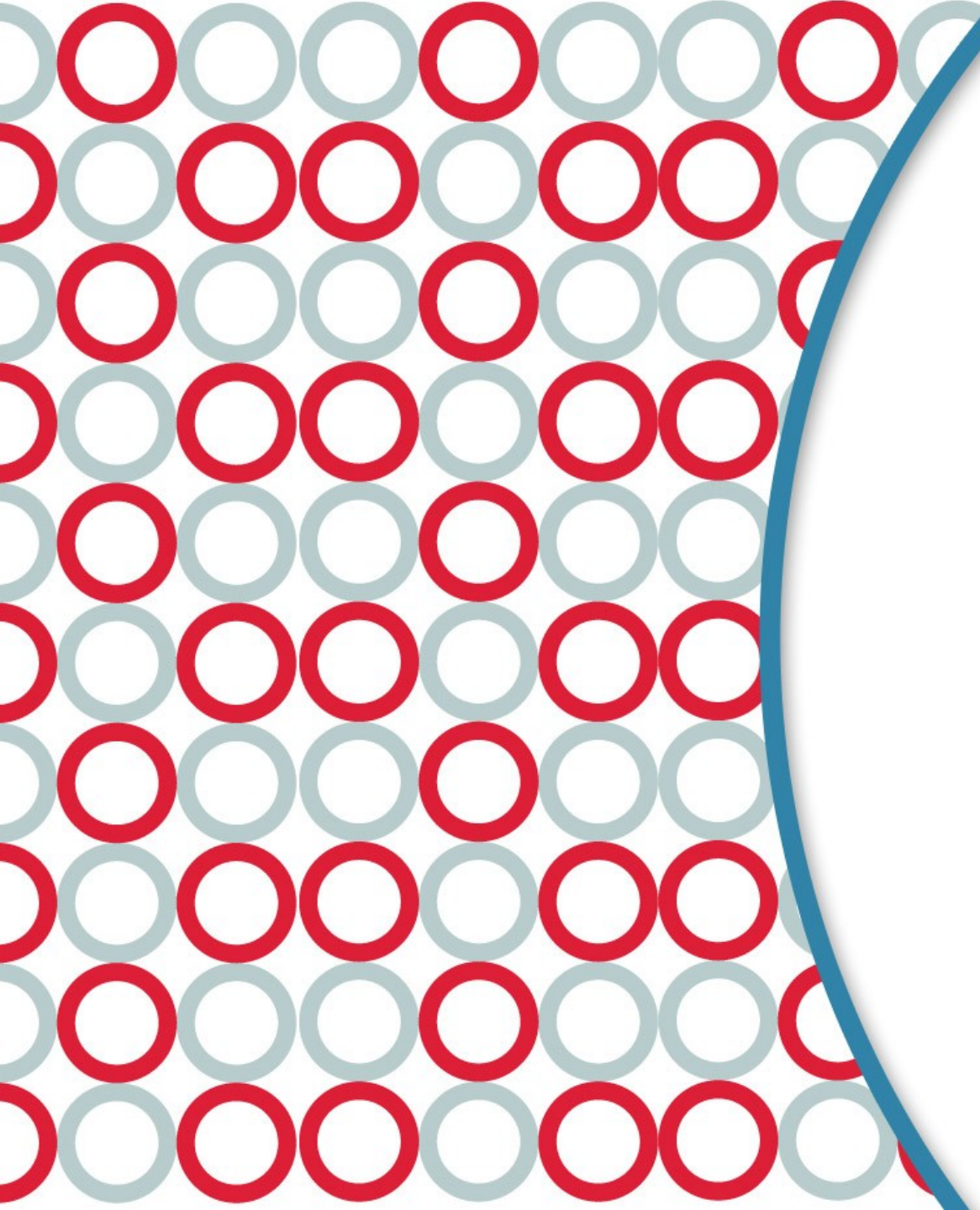


May 2019, IPT Course
Java Web Debelopment

JavaBeans

Trayan Iliev
tiliev@iproduct.org
http://iproduct.org

Where to Find the Code?

Intarmediate Java Programming projects and examples are available @ GitHub:

https://github.com/iproduct/course-java-web-development

Component Based Engineering

- Do Components Exist? [http://www.c2.com/cgi/wiki?DoComponentsExist]
- *They have to exist. Sales and marketing people are talking about them.* Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy. - Ralph Johnson

What is Software Component?

- Instead of extensive coding → components assembly into sub-systems and systems → components are like snapping blocks in Lego.
- "Snapping" between components is based on well defined interfaces and conventions.
- Separation of concerns – component developers code components, while application programmers assemble and configure them – preferably in a declarative way.
- Visual programming allows to greatly simplify the task of introspecting the components and expose/modify their configuration – example: Delphi.

Component Model:
”Write once use everywhere”

- Components are self-contained elements of software that can be controlled dynamically and assembled to form applications. ... These components must also interoperate according to a set of rules and guidelines. They must behave in ways that are expected. It's like a society of software citizens. The citizens (components) bring functionality, while the society (environment) brings structure and order.
- JavaBeans is Java's component model. It allows users to construct applications by piecing components together either programmatically or visually (or both). Support of visual programming is paramount to the component model; it's what makes component-based software development truly powerful.

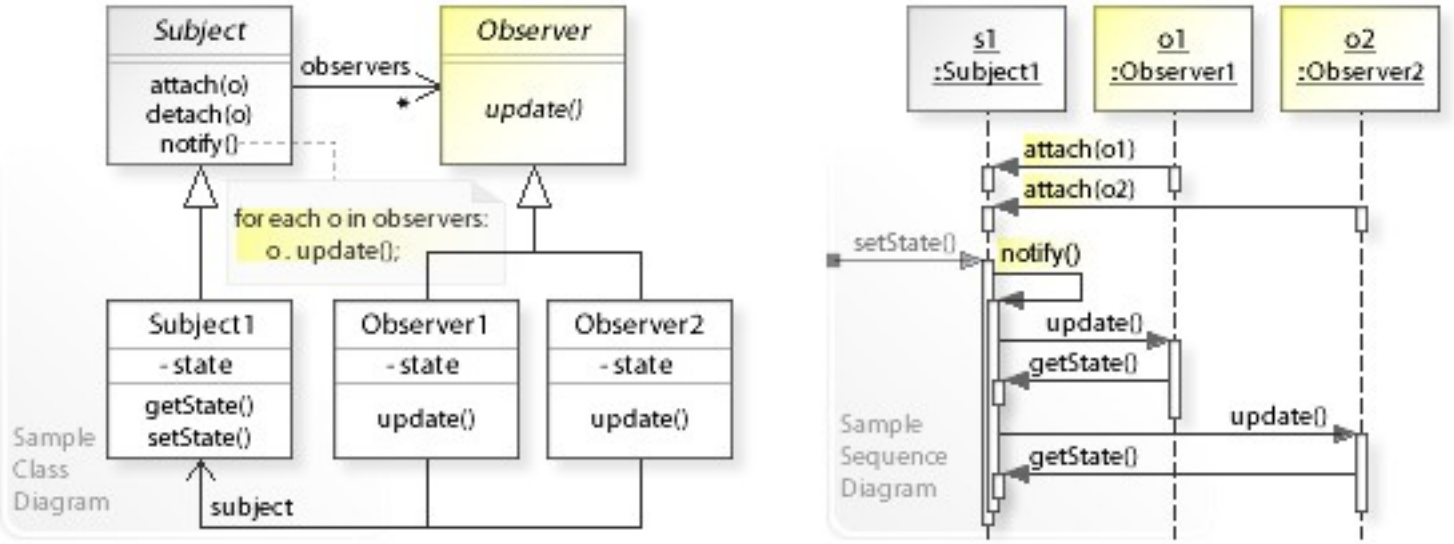
Robert Englander - “Developing Java Beans”

Component Registration and Discovery

- Components provide necessary metadata (specification) about their supported interfaces (contracts) which allows their dynamic discovery and assembly at runtime.
- There is no dependency on concrete component implementations (only on supported interfaces), which allows to be developed in parallel with their clients, and to be swapped with improved implementations without changing the existing code of client components.
- This is usually done using the principle of Inversion of Control (IoC) implemented either as Dependency Injection (DI - e.g. CDI) or a programmatic lookup (e.g. JNDI).

JavaBeans Component Aspects

- Event handling – Observer pattern



- State persistence – ***Serializable***, ***Externalizable***, customized
- Visual presentation, configuration, assembly
- Components and containers, layout, custom property editors

Other JavaBeans Aspects

- Design time vs. run time component handling – design time component provides information to the assembly tool about properties, methods and events, allowing to configure the properties and to write code for event handling (inter-component wiring). In runtime this allows appropriate component interaction directed towards achieving the system design goals.
- Visible vs. no visible components – design time props should be visible to the assembly tool, runtime visibility is optional
- Multi-threading – JavaBeans should be thread-safe
- Security – untrusted environments, be careful in assumptions
- Naming conventions (patterns) – optional, can use ***BeanInfo***

JavaBeans Naming Convention

- For each property **xxx**:
  - **getXxx( )**
  - **setXxx( )**
- For boolean properties "**get**" -> "**is**"
- For all public methods – exposed "as is" (bean business methods)
- For events **XxxEvent**:
  - **addXxxListener(XxxListener)**
  - **removeXxxListener(XxxListener)**

Examples

- Properties:

```
PropertyType propertyName;                    // declaration
public PropertyType getPropertyName() {/*...*/}     // getter
public void setPropertyName(PropertyType p) {/*...*/} // setter
```

- Events:

```
public void addXxxListener(XxxListener l) {/*...*/}
public void removeXxxListener(XxxListener l) {/*...*/}
```

JavaBeans - Example Bean

```
public class FaceBean extends JComponent {
    private int mMouthWidth = 90;
    private boolean mSmile = true;

    @Override
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        // Draw face ...
    }

    public int getMouthWidth() {
        return mMouthWidth;
    }
    public void setMouthWidth(int mw) {
        mMouthWidth = mw;
        repaint();
    }
    public void smile() {
        mSmile = true;
        repaint();
    }
    public void frown() {
        mSmile = false;
        repaint();
    }
}
```

Component Introspection & Customization

- Component introspection is the process of dynamic discovery of all JavaBeans properties, business methods, and supported events in runtime, by using the ***java.beans.Introspector*** class.
- When the automatic bean introspection is not enough, there is a customization mechanism: providing own component descriptor class implementing the ***java.beans.BeanInfo*** interface.
- IDEs provide UI for JavaBeans properties customization, providing a properties sheet built dynamically using introspection. You can provide custom ***PropertyEditor****s* by subclassing the ***PropertyEditorSupport*** class or even ***java.beans.Customizer*** implemetations such as wizards.

Interface BeanInfo

- **BeanInfo[] getAdditionalBeanInfo() - This method enables the current BeanInfo object to return an arbitrary collection of other BeanInfo objects that provide additional information about the current bean.**
- **BeanDescriptor getBeanDescriptor() - the bean descriptor that provides overall information about the bean, such as its display name or its customizer.**
- **int getDefaultEventIndex() - default event typically applied used.**
- **int getDefaultPropertyIndex() - A bean may have a default property commonly updated when this bean is customized.**
- **EventSetDescriptor[] getEventSetDescriptors() - event descriptors of the bean that define the types of events fired by this bean.**
- **Image getIcon(int iconKind) – an image that representing the bean**
- **MethodDescriptor[] getMethodDescriptors() - Returns the method descriptors of the bean that define the externally visible methods supported by this bean.**
- **PropertyDescriptor[] getPropertyDescriptors() - Returns descriptors for all properties of the bean.**

Extracting BeanInfo using Introspector (1)

```
BeanInfo bi = null;
try {
  bi = Introspector.getBeanInfo(bean, Object.class);
} catch(IntrospectionException e) {
   print("Couldn't introspect " +  bean.getName()); return;
}
for(PropertyDescriptor d: bi.getPropertyDescriptors()){
  Class<?> p = d.getPropertyType();
  if(p == null) continue;
  print("Property type:\n  " + p.getName() + ", name:\n  " + d.getName());
  Method readMethod = d.getReadMethod();
  if(readMethod != null) print("Read method:\n  " + readMethod);
  Method writeMethod = d.getWriteMethod();
  if(writeMethod != null) print("Write method:\n  " + writeMethod);
}
```

Extracting BeanInfo using Introspector (2)

```
print("Public methods:");
for(MethodDescriptor m : bi.getMethodDescriptors())
  print(m.getMethod().toString());
print("Event support:");
for(EventSetDescriptor e: bi.getEventSetDescriptors()){
  print("Listener type:\n  " + e.getListenerType().getName());
  for(Method lm : e.getListenerMethods())
    print("Listener method:\n  " + lm.getName());
  for(MethodDescriptor lmd :
      e.getListenerMethodDescriptors() )
    print("Method descriptor:\n  " + lmd.getMethod());
  Method addListener= e.getAddListenerMethod();
  print("Add Listener Method:\n  " + addListener);
  Method removeListener = e.getRemoveListenerMethod();
  print("Remove Listener Method:\n  "+ removeListener);
```

Property Editors

```
public Object getValue();
public void setValue(Object value);
public String getAsText();
public void setAsText(String text);
public boolean isPaintable();
public void paintValue(Graphics g, Rectangle box);
public boolean supportsCustomEditor();
public Component getCustomEditor();
```

Specifying Property Editor & Customizer

```
public void paintValue(Graphics g, Rectangle box) {
     Color oldColor = g.getColor();
     g.setColor(Color.BLACK);
     g.drawRect(box.x, box.y, box.width - 3, box.height - 3);
     g.setColor(color);
     g.fillRect(box.x + 1, box.y + 1, box.width - 4, box.height - 4);
     g.setColor(oldColor);
}
```

```
// Specify the target Bean class, and,
// If the Bean has a customizer, specify it also.
private final static Class beanClass = BulbBean.class;
private final static Class customizerClass =
                                        BulbBeanCustomizer.class;
public BeanDescriptor getBeanDescriptor() {
       return new BeanDescriptor(beanClass, customizerClass);
}
```

Property Change Listeners

```
// Property change support for bound property
private PropertyChangeSupport changes = new
PropertyChangeSupport(this);

public void addPropertyChangeListener(PropertyChangeListener lis) {
     changes.addPropertyChangeListener(lis);
}

public void removePropertyChangeListener(PropertyChangeListener lis) {
     changes.removePropertyChangeListener(lis);
}

public void setClosed(boolean newStatus) {
     boolean oldStatus = closed;
     closed = newStatus;
     changes.firePropertyChange("closed", new Boolean(oldStatus), new
Boolean(newStatus));
}
```

Bean Serialization

- Interface ***Serializable*** – all fields are serialized automatically, except those declsared as ***transient***
- Интерфейс ***Externalizable*** – we serialize everything explicitly
- Implementing private methods ***readObject()*** and ***writeObject()*** – *Serializable* with customization when necessary
- Implementation examples

JavaBeans Support

- Class (java.lang.Class)
- Class Loader: java.lang.ClassLoader, java.net.URLClassLoader
- Package: java.lang.reflect
- Package: java.beans
  - Introspector: java.beans.Introspector
  - Feature Descriptors: BeanDescriptor, EventSetDescriptor, MethodDescriptor, PropertyDescriptor, ParameterDescriptor.
  - Bean Utilities: java.bean.Beans
  - Property change and vetoable change:
  - BeanInfo:
  - Editor:
  - Customizer:
  - Serialization: XMLEncoder, XMLDecoder.

References

- Trail: JavaBeans(TM) - https://docs.oracle.com/javase/tutorial/javabeans/index.html

- Javabeans Tutorial:

https://www.ntu.edu.sg/home/ehchua/programming/java/J9f_JavaBeans.html

Thank's for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg